Københavns Universitets Matematiske Institut
Afdelingen for Informationsbehandling
Universitetsparken 5.

IMP, a Programming Language, with floating Definitions

by

Bj. Svejgaard

K.U.A.I.B. 15

IMP, a Programming Language, with floating Definitions.

The title of this lecture may in fact be a little misleading. It tells the truth indeed, and nothing but the truth, but not the whole truth.

The language, I am going to speak about to day is much more than a programming language. It may be used for programming purposes, and it very often is. But it may be used for so many different purposes outside the field of programming that it would not be doing justice to the language, merely to call it a programming language.

To justify the structure of the language I should like to put forwards my opinions about automatic computers and the general way of constructing complicated logical concepts.

In nearly all languages the word used to denote automating data processing equipment contains the notion of computing. Unfortunately enough this fact stresses a single feature of the machine.

It is true that the use made of computers are to a large degree numerical. However to explain computers fully we must realise that the central mechanism of computers is far from being numeric. Perhaps a computer may best be described as an information transforming automaton. That is to say, that what the computer really can do is to accept a sequence of signals which we normally will call symbols. These symbols may be signals recieved from photocells, relays or they may be signals

originating from the reading of punched cards or punched paper tape, signals produced by pressing a key on an electrical typewriter, and so on.

In the same way the computer may give out a sequence of signals resulting in holes in a card, in a paper tape, in a movement of the pen on curve plotter.

The fundamental capability of a computer is the transformation of a recieved string of symbols into another string of symbols according to a set of rules that are normally referred to as the programme. In this connection it is our main interest that any finite set of elements may be represented by the elements of a finite set of symbols consisting of not less than two elements. This fact signifies that the hardware symbols at any computer installation may be used to represent any string of symbols, that is, any kind of information.

The work done in any computer installation is just supplying a computer with strings of symbols. Sometimes these symbol strings are what we call programmes and sometimes we call them data, but there is no fundamental difference between them. Let us assume that we have a compiler for the translation of some autocode to a certain machine language. The string of symbols representing the programme in the source language is then data for the compiler, whereas the translated string later will be regarded as the programme.

The string of input symbols may be the necessary data for the integration of some differential equation, in which case the

output string will be what we call the solution of the problem. The input string may also be one of Shakespears plays and the output string may be its translation into Greek. It seems to me that the most general view of the working computer will be about the following:

1) The computer reads a string of symbols currently.
2) During the reading the machine produces a transformed string which partly may be stored inside the computer.
3) The transformation will take place according to the rules layed down by the transformed string that is at any moment inside the computer.

From this follows that every problem in principle may be solved by some string producing mechanism that is by some language.

If this language is going to be used as generally as already mentioned, it is very important that the language itself should be quite independent of the use to be made of the resulting string. That is, the processor should be independent of the target language. If this principle is maintained it is quite clear that such a language may be used as a programming language, and it may be so for any machine and for any target language on any other machine. But even when we are programming for a specific machine the general mechanism may be very useful.

If we regard the usual situation for a programmer, we will see that he is faced with five possibilities of programming.

He may use

1) a simple numerical code,
2) a symbolic code with mnemonic operation codes and symbolic addresses,
3) an assembly language with possibility for macroinstructions,
4) an autoprogramming language such as Algol, Cobol, Fortran etc.,
5) special languages pertaining to a special programming method, e.g. LISP, IPLV.

Of these five possibilities the simple numerical code is so primitive that nowadays it cannot seriously be considered for more advanced programming.

The symbolic code, possibly combined with an assembly language, is the natural means for controlling a machine in all details and will often be used for programming of problems of a non-numerical type.

The languages of a more problem oriented nature will be used for describing a process and leave its actual execution to the compiler.

In spite of the greater programming effort an experienced programmer will normally resort to symbolic programming languages because of the more extended freedom and greater possibilities of that kind of language, and because he knows, that each problem really demands its own language.

What he really needs is a full spectrum of programming langu-

ages in which all degrees of freedom and all degrees of automatism are present or could be included, according to the programmers wishes. When we got so far in our reasonning in the department for information processing at the University of Copenhagen, we decided to make a language that was so general that the programmer would get the possibility himself of using it to all degrees of sophistication, be it in the same or in different programmes.

We named it IMP that is Interpretive Macro Processor and we call it a macro language. However it is not a macro language in the usual sense of this word. An ordinary macro language will accept macro definitions of a certain format and translate them into machine instructions for a specific machine. IMP is a string producing language and the produced string may be anything the user likes it to be. That is, it may be a programme in any of the already mentioned types of programming languages either for the computer really used or for another machine. It may be used for writing file cards, for filling in forms, for writing business letters and so on.

If such a language should be manageable it is very important that its structure is rather close to the normal way of constructing concepts.

The usual way of doing this is to construct from some basic elements more complex sets. These sets we will give names and use them as new elements for constructing still more complex sets.

For programming purposes, however, it is essential, that we may execute this process in the opposite direction too. That is, we want to construct a rather complex entity, which is our programme, and we want to construct it from elements taken to exist already. Later we go on defining these elements by still simpler entities and so on, until we eventually reach the basic elements which are themselves elements of what we call target language.

The syntax of the IMP language is so simple that it may be written down here. Among the symbols available in the hardware system we pick out a number, in the present form of the language 11 of them.

We may represent them here as

[ ] ( ) * = c r e ≤ ^

These will be delimiters of the language. The two first are used for the macro definition. It is not important what characters are used as delimiters, because we may at any time redefine them. When the last of the selected delimiters are met, the processor will take the next eleven characters as new delimiters.

All the remaining characters in the hardware system may be used freely. Any sequence of them is called a basic string.

<definition>::= [<name>*<comp.string>]

<name>::= <basic string>

<basic string>::= <symbol>|<basic string><symbol>|<empty>

<comp.string>::= <basic string>|<definition>|<call>|<formal>|

<comp.string><comp.string>

<call>::= (<name><list>)

<list>::= <actual>|<list><actual>

<actual>::= ¥<comp.string>

<formal>::= =<pos.integer>=

It follows that several of the definitions are recursive.

This may all look a little complicated, but it is really simplicity itself.

A few example will serve to illustrate the concepts:

A usual abreviation is really what we here have named a macro. Take e.g. the word SABENA. It stands for
Societe Anonyme Belge pour l Exploitation de la Navigation Aerienne.

In IMP this may be expressed by writing
[SABENA¥Societe Anonyme Belge pour l Exploitation de la Navigation Aerienne]

Whenever we write (SABENA) this will be replaced by the long string by the processor. If we want the string SABENA itself, we just write SABENA.

This would not be very useful were it not for the parameters. In the definition of a macro we may put in as many formal papameters, as we like.
When the macro is called the formal parameters will be replaced

by the actual parameters having the same number. The actual parameters will again be strings probably written in the same way and so on to any number of levels.

The concept of locality applies in the same way as in Algol, the definition brackets having the same effects as begin and end in Algol.
The following example shows a possible IMP string. For pedagogical reasons it has been chosen quite nonsensical just to underline the fact, that the language will handle strings of symbols without any regard of their meaning.

```
(M) i (M)
(P*ab*p)
[M*if not][P*AUT=2=F
NIM=1=.]
[a+b*123(M)910=3=f[M*ttt]]
(a+b*1*4*A(M)B*AB)]
```

will result in the following string

```
if not i if not
AUTpF
NIMab.
123ttt910Aif notBf
```

First the macro M is called twice with an intervening string consisting of one single i. The macro M has not yet been defined, but this is quite permissible.

The macro P is called still without being defined.

Now the macro M is defined. It is defined as the string if not without any parameters. The macro P is defined as the indicated string with two parameters.

The definition of the macro a+b is more interesting. We see that a call of a macro M occurs in the definition af a+b. Therefore this local definition is used inside a+b.

When the macro a+b is called, the definition of M inside a+b is inaccessible, so the meaning of M in this place must be if not. We see furthermore that a+b has one formal parameter having the number 3. In the call this will be replaced by the actual parameter having 3, that is, A(M)B with the meaning Aif notB. This shows that a call of a macro may contain more actual parameters than there are formal parameters in the definition of the macro.

In Algol we may use procedures for facilitating the writing of a programme. In case of most compilers we are faced with the fact, that if the amount of processing to be done by the procedure is small we may loose time because of the time taken by the procedure call itself.

If computing time is essential, the obvious thing to do is to replace the procedure call with the procedure body .

In this case it would be better to replace the procedure definition by an IMP macro, and to replace the procedure call by

the call of the macro.

The effect will be that wanted for. The programming effort will be exactly the same as if we had used procedure, but the programme will look and work exactly as if we had written the procedure body in all places.

E.g. the multiplication of two complex numbers may be executed by the piece of programme generated by the following macro.

```
[cm*_
=3=[0]:= =1=[0]×=2=[0]-=1=[1]×=2=[1];
=3=[1]:= =1=[0]×=2=[1]+=1=[1]×=2=[0];]
```

which may be called e.g. as (cm*p*q*r).

But we may do even better than this. If the following macro definitions are included in an algol programme,

```
[r*=1=[0]][i*=1=[1]]
[mu*
au:= rp×(r*=1=)-ip×(i*=1=);
ip:= rp×(i*=1=)+ip×(r*=1=); rp:= au;]
[ad*
rp:= rp+(r*=1=); ip:= ip+(i*=1=);]
[in*
rp:= (r*=1=); ip:= (i*=1=)]
[out*
(r*=1=):= rp;  (i*=1=):= ip;]
```

the calls

(in*c2)(mu*z)(ad*c1)(mu*z)(ad*c0)(out*w)

will produce a string, that is a set of Algol statements necessary to compute and store the value of a complex quadratic polynomial. (We have used slightly altered delimiters to avoid confusion with those of Algol).

```
rp:= c2[0]; ip:= c2[1]
au:= rp×z[0]-ip×z[1];
ip:= rp×z[1]+ip×z[0]; rp:= au;
rp:= rp+c1[0]; ip:= ip+c1[1];
au:= rp×z[0]-ip×z[1];
ip:= rp×z[1]+ip×z[0]; rp:= au;
rp:= rp+c0[0]; ip:= ip+c0[1];
w[0]:= rp;  w[1]:= ip;
```

This principle may of course be extended to a full complex arithmetic.

We see that the real part and the imaginary part of a complex variable have been called by a macro too. This will be useful if we choose to use e.g. the indices 1 and 2 instead of the indices 0 and 1 for the two parts. In this case we must just redefine the macros r and i, but the rest of the programme will still be applicable. The example also shows that an actual parameter for one macro may be formal for another.

The macros r and i could also be defined as [r*-1-0]] and [i*-1-1]], which will permit calls as (r*c[2,), (i*z[) giving

respectively c[2,0] and z[1]. This should again underline the fact, that the IMP language need not handle syntactical entities in any target language.

If we note the sequence (mu*z)(ad... we should define a compound operation by [muad*(mu*=1=)(ad*=2=)]. The calls (in*c2)(muad*z*c1)(muad*z*c0)(out*w) will then produce the same Algol string. We see that Algol programme is not quite efficient, but we could improve it by using the last calls if we define

```
[muad*
au:= (r*=2=)+rp×(r*=1=)-ip×(i*=1=);
ip:= (i*=2=)+rp×(i*=1=)+rp×(r*=1=);
rp:= au;]
```

I have used Algol as the target language because I assume that it is the most commonly known language.

We could however redefine all the macros in a suitable machine language, and the same calls may still produce the same effect.

This is what we mean by using the word floating definition. In this way it very often has happened that we have totally rewritten a programme without doing anything else than redifining a set of macros.

The difference between an IMP macro and an ordinary macro instruction really comes into play when we use IMP for facilitating the writing of machine programmes.

I should, however like first to take an example where the macro is used to generate a string which will be instructions for a computer.

In the GIER computer the following commands will calculate the value of a polynomial, the coefficients of which are stored consecutively,

panr1t=1=

arX1

qqVXLA

mk=2=, hvr-2

where =1= is the address of the highest term coefficient and =2= the address of the argument.

If we take exactly that string of symbols to define a macro poly with two formal parameters, then we can programme by just writing (poly*(alpha)*(time)) where (alpha) is the name of the coefficient array (or rather the macro defined as the address of its first location) and (time) is the name of the argument.

In this way we could do all computations of polynomials by means of open subroutines. It is a fact, that the use of macros will tend to increase the use of open subroutines instead of closed ones. Closed subroutines are used for two reasons. One is to save place in the memory, the other is to save programming effort.

But if the necessary space is available, the open subroutine is to be preferred, because normally it will not demand so much administration of the parameters as the closed subroutine, and so they will usually be faster. The saving of programming effort

is just as well maintained by using macros.

If space is getting sparse during programming one can always redefine the macros used for open subroutines as jumps to closed ones.

E.G. in our case we could define

```
[poly*hs(polynomial subroutine)
 qq=1=t=2=]
```

where hs is the jump instruction and the next command represents a packing of the parameters.

A perhaps more typical use of the macro for machine programming is shown by the following simulation of the FORTRAN if statement.

```
[IF*=4=
 hvr=1= LT
 hvr=2= LZ
 hvr=3=]
```

The three commands will jump to the addresses given by the parameter =1=, =2=, =3= if the accumulator is negative, zero or positive. The parameter =4= will represent a set of commands the result of which will decide which of the three following commands should be effective.

E.g. the call

```
(IF*a1*b2*c1*arn(alpha)]
```

will cause jumps to one of the labels a1,b2 or c1 according to the sign af (alpha), arn being the instruction for clearing the accumulator and adding something to it.

The difference between an ordinary macro and a macro in the IMP language is even more conspicious in the following example.
We define in the symbolic language of the GIER computer

```
[loop*be1
 gpr2, pp
e:it(=1=),bsp1
 pp, hvre1=2=
 ppp1, hvre
 e1=i
 e]
```

the underlined b and e has the same meaning as begin end in Algol. Two local addresses e and e1 are declared by writing e1 after the b.
The commands in the first line of the macro will store the contents of the index register in the address part of the third line and set the index register (p) to zero. The commands having the label e will test if the contents of the cell with the address given by the parameter =1= is larger than the contents of the index register. As long as this is the case the instruction in the next cell will not be executed, but the machine will execute the instructions given in the parameter =2=.
The commands ppp1, hvre will advance the index register by 1 and jump to the label e. When the condition at this location is no longer satisfied the instruction pp, hvre1 will restore the original value of the p register and jump to the label e1.
By this organization it is possible to have any number of loops inside each other without using more than one index register.

E.G. the calls

```
(loop*(alpha)*
 ............
(loop*(beta)*
 ...........
 ...........
 ...........))
```

will do the same work as the Algol statement

```
for i:= 1 step 1 until alpha do begin
........
for j:= 1 step 1 until beta do begin
........ end;
........ end;
```

If-then-else constructions could be defined in a similar way. Here again I have used the symbolic language of the GIER computer as a target language, but that is of course not important, as a similar machanism may be defined in nearly any other machine language. In this way the programmer may be able to define macros doing most of the work of the standard mechanisms in e.g. Algol and use them when he likes. And what is more important, he may omit the use of them if he prefers so.

It may be said, that all that kind of things could be done without IMP, and be replaced by a set of rules. That is correct. From this viewpoint we could say that IMP is nothing but a mean for making the computer use a set of rules.

But this is just the point where the computer is superior to man. The machine is able to stick to even a large set of complicated rules, and to apply them, whereas man is not. In fact

it has been possible by means of IMP to construct machine programming languages with automatic dynamic storage allocation dynamic and static stack etc.

To take a last example, this time outside the field of programming, let us assume, that we have a list of data for several persons.

family name
first name
address
telephone number
car number
weekly salary
extra holyday salary
subsidiary information
(date)

We may make this the call of a not yet defined macro, say M by putting [M at the front and] at the tail. If we have chosen carriage return (new line) as the parameter delimiter.
This list of calls may be used for many different purposes just by defining the macro M differently in each case. If e.g. we want to mail the extra holydays salary to all persons on the list, we must use a certain mail form.

In this we will have to fill in the recievers name, address, the amount of money, and the data, each of them in three places.

In this case we will just define the macro M by filling in the

parameter numbers in the appropriate places like this:

```
[M
=2==1=          =2==1=          =2==1=
=3=             =3=             =3=

=6=             =6=             =6=
                                =9=
]
```

and define date by e.g.

```
[date
 4. jan 1966],
```

then the IMP will do the rest.

One may think that a language like this should demand a very complicated processor. This is however not the case.

Interpretive Macro Processor for the Gier takes up 680 words of memory. The processor was ready early in the beginning of 1964. It has turned out to be a very powerfull tool for nearly all work in information processing.

Our working with the language has confirmed our opinion, that the fewer restrictions in the language the more powerful it will be.

One thing which can be done and will be done is to permit calls of macros in a macro name, that is, the syntax will be changed to the following

<definition>::= [<name>*<string>]

<name>::= <string>

<string>::= <symbol>|<definition>|
<call>|<formal>|<string><string>

This change will allow for conditional macros too. A language of a similar type the General Purpose Macroprocessor had recently been published by C. Strachey of the Massachusetts Institute of Technology.

For years now a very large work has been done to supply ordinary users of computers with an easy programming language. But so far the task has been neglected to supply the experienced programmer with a time-saving tool that will deprive him of no facilities of his computer.

To my opinion therefore languages of the IMP type will be an essential part of the future software system of any computer.

Bj. Svejgaard.